TOWER

Outline/draft of Gawk manual

Paul Rubin

LAST TIME WRITTEN: Wed Mar 4 18:26:24 1987

Copyright © 1986, by Paul Rubin. This will be free documentation eventually, but not until it is finished.

Table of Contents

1. What **Awk** is about 1
1.1 Brief summary of **awk** 1
1.1.1 When not to use **awk** 1

2. Awk Programming 3
2.1 Patterns 3
2.1.1 The Empty Pattern 3
2.1.2 Regular Expressions as Patterns 4
2.1.3 Conditional Expressions as Patterns 4
2.1.4 Specifying Line Ranges with Patterns 4
2.1.5 **BEGIN** and **END** 5
2.2 Actions 5
2.3 The Data Types of **Awk** 6
2.4 **Awk** Variables and Arrays 7
2.5 Expressions 8
2.5.1 String Operations 8
2.5.2 Numeric Operations 9
2.5.3 Assignments 10
2.5.3.1 The ++ and -- operators 11
2.5.4 Fields 11
2.5.4.1 Accessing Fields 11
2.5.4.2 Assigning to fields 12
2.5.4.3 Specifying how fields are separated 13
2.5.4.4 Choosing your Separator Character 13
2.6 Conditional Expressions 14

3. Control Statements 16
3.1 The **if** statement 16
3.2 The **while** statement 16
3.3 The **for** statement 16
3.3.1 Looping Through an Array 18
3.4 The **break** statement 18
3.5 The **continue** statement 19
3.6 The **next** statement 19
3.7 The **exit** statement 19

4. Input and Output 21
4.1 Input 21
4.2 The **Print** and **Printf** commands 21
4.2.1 Redirecting output of **print** and **printf** 22

Appendix A. Builtin Functions 24

Appendix B. Special Variables 26

Appendix C. A Text Analysis Example 27
C.1 More examples 28

Appendix D. Implementation notes and bugs 29
D.1 Known bugs and misfeatures in **gawk**. 29
D.2 How **gawk** works inside 29
D.3 Improvements: suggestions for future improvements 29
D.4 Differences: differences from Unix awk 30

Appendix E. Glossary 31

1. What Awk is about

1.1 Brief summary of awk

Awk is a pattern scanning and string processing language that allows you to do fairly elaborate processing of fixed-format input files with little programming effort. For example, the command

```
ls -l | awk '$5=="Nov" {sum += $3} \
              END {print sum}'
```

prints the total number of bytes in all the files in the current directory that were last modified in November (of any year).

Awk can be thought of as a sophisticated stream editor. This means that **awk** programs process a single input stream (either the input or a sequence of files) and usually write a single output file (although features are provided to write output onto any file or pipe). **Awk**'s name comes from the initials of its original designers (Aho, Weinberger, and Kernighan) who also wrote the Un*x implementation. The GNU implementation **gawk** was written by Paul Rubin and Jay Fenlason, with advice from Richard Stallman. Jay Fenlason, Richard Mlynarik, and Richard Stallman gave helpful comments on drafts of this manual; and the paper *A Supplemental Document for* **Awk** by John W. Pierce of the UC San Diego Department of Chemistry pinpointed several issues relevant to both **gawk**'s implementation and to this manual that would otherwise have escaped us.

Whenever we specifically refer to **gawk** in this manual, we do so to distinguish some aspect of its behavior from other implementations of **awk**, or because we are describing the method **gawk** uses to do something that other implementations might do in a different way. Also, where it will not cause confusion, we begin every sentence with an upper-case letter even when the word thus capitalized is an **awk** keyword which in an **awk** program would have to be typed in all lower case.

1.1.1 When not to use awk

Awk is very useful for simple batch editing tasks, like summarizing information from the output of standard programs such as **ls**. More powerful systems have also been written in **awk** including a complete retargetable assembler for 8-bit microprocessors and a microcode assembler for a special purpose Prolog computer. However, **awk**'s capabilities are rather strained by tasks of that much complexity. If you find yourself writing **awk** scripts of more than, say, a few hundred lines, you

might consider using a different programming language. **Icon** is probably a good choice if you need sophisticated string or pattern matching capabilities. More conventional languages like C or Lisp offer better facilities for system programming and for managing the complexity of large programs. Programs in these languages may require more lines of source code than the equivalent **awk** code, but they will be easier to maintain and usually run more efficiently.

2. Awk Programming

An **Awk** program (or *script*) consists of a (possibly empty) series of *rules*, where a rule is a *pattern* accompanied by a fragment of **awk** code (or *action*). When an **awk** script is executed, the input given it is read one line at a time. Then, for each rule in the script, that rule's action is performed if the rule's pattern is satisfied by the input line. Each line can run several rules.

Patterns are arbitrary conditional expressions involving numerical, boolean, and string comparisons and matching regular expressions against strings. Actions are written in a language with C-like syntax that can use **awk**'s features for string manipulation, extracting fields from input records, etc.

Example: the rule

```
/foo/ { ++foobar }
```

preincrement operator 2.5.3.1 p. 11

contains the pattern **/foo/** along with the action **{ ++foobar }**. The pattern is satisfied whenever an input line contains the string '**foo**', and the action increments the variable **foobar**. Thus, this rule counts how many lines in the input contain the string '**foo**'. A pattern can also specify *ranges* of input lines to process. Complete information on patterns is given in the next section. For information on actions, see <undefined> [Actions], page <undefined>.

2.2 5

2.1 Patterns

A pattern can be of the following forms:

null The empty pattern

condition A single conditional expression

condition1, *condition2*
A pair of conditional expressions separated by a comma, specifying a range.

BEGIN or **END**
The special patterns **BEGIN** or **END**.

2.1.1 The Empty Pattern

An empty pattern is considered to match *every* input line. Thus, the rule

```
# a less boring example than this would be nice here
{ nlines++ }
```

counts the total number of lines in the file and keeps the count in the variable **nlines**. The special variable **NR** does the same thing (see <undefined> [Special Variables], page <undefined>) making this a useless rule, but it illustrates the use of an empty pattern.

2.1.2 Regular Expressions as Patterns

If you specify a *regexp* as a pattern, the pattern will be satisfied by exactly those input lines which match the regexp.

Blah, blah, blah for another paragraph or so.

2.1.3 Conditional Expressions as Patterns

If the pattern is a single conditional expression, the expression is evaluated for each input line read and the pattern is satisfied if the condition is true. Thus, the rule

```
$1=="foo" { print $2}
```

prints the second field of each line whose first field is the string '**foo**'. See <undefined> [Fields], page <undefined> for more information about fields.

2.1.4 Specifying Line Ranges with Patterns

If the pattern is two conditional expressions separated by a comma, it specifies *ranges* of lines in which to execute the rule's action. Ranges in **awk** patterns seem to have been inspired by the old Un*x line editor **ed**. They work as follows: *condition1* is evaluated for every line in the input until it is satisfied. When *condition1* is finally satisfied, the pattern becomes "turned on". When a pattern is turned on, it matches every input line read, up to and including the next one satisfying *condition2*. Then it becomes turned off again until another line satisfying *condition1* is read. It is possible for a pattern to be turned both on and off by the same line, if both conditions are satisfied by that line. Then the action is executed for just that line.

Example:

```
$1=="on",$1=="off" { print $0 }
```

prints every line between on/off pairs, inclusive. (The field specification `$0` refers to the whole line. See <undefined> [Fields], page <undefined> for an explanation of fields.) A null action could also have been used for this.

2.1.5 BEGIN and END

`BEGIN` and `END` are special patterns used for supplying start-up or clean-up code to your **awk** script. `BEGIN` is only satisfied before the first input record has been read and `END` is only satisfied after end-of-file on all the input files has been reached. That is, `BEGIN` and `END` test for beginning- and end- of input, respectively.

A complete **awk** program that uses `END` is the following:

```
/foo/ {++foobar}
END { print foobar }
```

This program consists of two rules. The first rule increments the variable `foobar` every time a line containing '`foo`' is read, because it has the pattern `foo` which is satisfied only by input lines containing the string '`foo`'. See <undefined> [Pattern Regexps], page <undefined> for more on this. The second rule prints out the value of `foobar` after all the input has been processed. Note that the shell command `grep foo | wc -l` would probably have run faster than the **awk** script given here.

2.2 Actions

Every **awk** action consists of one or more **awk** *statements*, enclosed in curly braces (`{` and `}`). The curly braces must be supplied even if the action contains only one statement. Statements are separated by newlines or semicolons.

There are three kinds of awk statements: expressions, control statements, and compound statements.

- *Expressions* include assignments and function calls as special statements. There are a lot of different kinds of operations and data objects that can be found in **awk** expressions; this manual has a whole section describing them. See <undefined> [Expressions], page <undefined>.

- *Control statements* specify the control flow of **awk** programs. The **awk** language gives you the usual C-like constructions (**if**, **for**, **while**, and so on) as well as a few special ones. See <undefined> [Control Statements], page <undefined>.
- A *compound statement* is just one or more **awk** statements enclosed in curly braces.

2.3 The Data Types of Awk

Every **awk** object has an associated *data type*, either *numeric* or *string*. Objects of numeric type are *numbers*.

Strings in **awk** can be of any length, and at least in **gawk** they can contain all the possible 8-bit ASCII characters including **NUL**. (This might not be true in Un*x **awk**.) **Awk** always represents numeric objects as floating point numbers. Unlike Un*x **awk**, **Gawk** normally uses double precision when storing or computing with floating point numbers (you can make **gawk** use single precision instead by changing a parameter set when the **gawk** sources are compiled).

If you use a data object of one type where the context calls for a different type, **awk** automatically converts (or *coerces*) between types for you. For example, if the values of **foo** or **bar** in the expression **foo + bar** happen to be strings, they are converted to numbers. Similarly, the statements

```
two = 2
three = 3
foo = (two three) + 4
```

set **foo** to the (numeric) value 27. (The numeric variables **two** and **three** are converted into strings and concatenated together, and the resulting string is converted back to a number before 4 is added.)

The exact manner in which numbers are converted into strings is controlled by the **awk** special variable **OFMT** (see <undefined> [Special Variables], page <undefined>, for more information about special variables). The numbers are converted using an internal version of the **sprintf** function with **OFMT** as the format specifier (see <undefined> [Printf], page <undefined>). **OFMT**'s default value is **"%.6g"**. You might want to change it to specify more precision, if your version of **awk** in fact does use double precision arithmetic. Double precision on most modern machines gets you 16 or 17 decimal digits of precision.

Strange results can ensue if you set **OFMT** to a string that doesn't tell **sprintf** how to format

floating point numbers in a useful way. For example, if you forget the `%` in the format, all numbers will be converted to the same constant digit string. If you set `OFMT` to a numeric value, **gawk** can't know how to convert it any more, so it uses the default value of `"%.6g"`.

A string that doesn't represent a well-formed number is converted by examining the longest leading substring it has which *does* represent a well-formed number. If no such substring exists, the string is converted to the number 0.

2.4 Awk Variables and Arrays

Awk has variables and arrays like other languages, but there are some unusual things about them. First, variables are dynamically typed and string storage is managed automatically. Second, arrays are *associative.* This means array subscripts don't have to be integers; they can have any values legal in **awk**, including strings and floating point numbers. **Awk** arrays are thus reminiscent of the **table** objects of SNOBOL4.

Names of variables and arrays are strings of alphanumeric characters and underscores beginning with an alphabetic character.

Examples of setting variables and elements of arrays:

```
x = 5
foo["xyz"] = "bar"
x[17] = 1.61808339
```

The first example here sets the variable `x` to the number 5. The value thus associated with the variable called `x` is completely independent from any values that might be stored in an array also called `x`.

The second example sets the array element `foo["xyz"]` to `"bar"`. The third example illustrates how a numeric expression can be used as a subscript, just like a string expression. (The variable `x` remains set to 5, as stated above.)

Array elements are accessed through a hash function which is reasonably fast. Multi-dimensional arrays are not supported, but you can fake multiple subscripts using concatenation (see <undefined> [String Operations], page <undefined>):

```
row = 3
```

```
row = 3
col = 15
matrix[row "," col] = 23
```

sets `matrix["3,15"]` to 23. What has happened here is that the numbers 3 and 15 have been coerced to strings (see <undefined> [Data Types], page <undefined>) and concatenated together with a comma between them before being used as the string-valued subscript `"3,15"`. The comma is needed to distinguish the subscripts that would be generated by `row = 3, col = 15` from those of `row = 31, col = 5`.

Note that `a[17]` and `a["17"]` are the same object. When you set an array element using a numeric valued subscript, first, **gawk** checks whether that element already has been set; if it has, the new value is stored in its place and everything is ok. If not, **gawk** next coerces the subscript to a string and checks again, and only if element element does not exist using either form of the subscript is a new element created. Your code will almost certainly break if, for example, you change the contents of `OFMT` in between uses of such subscripts because the conversions will then have different results. See <undefined> [Data Types], page <undefined> for more about how type coercion works. It is not clear why **awk**'s original designers decided that subscripts should be converted like this. **Gawk** uses this method in order to be compatible with Un*x **awk**.

In order to examine *all* the elements of an array, you must use the `for (x in ...)` statement. See <undefined> [Array For], page <undefined>.

2.5 Expressions

Awk expressions include assignments, arithmetic and string operations, and function calls. Most operations try to work on every kind of **awk** data object. Most expressions have *values*, upon which further computation can be done. The exception is *conditional expressions*. Conditional expressions are special expressions which test relations between objects. Instead of having values, they direct the **awk** interpreter to execute or not execute given pieces of code.

2.5.1 String Operations

The most fundamental string operation is *concatenation*. Concatenating two strings together means sticking them together, one after the other. In **awk**, concatenation is specified by giving the two operands, one after the other, with no operator at all between them. This is similar to the way concatenation is specified in SNOBOL.

An example of concatenation: executing the statements

```
a = "foo"
b = "bar"
c = a b
```

sets the variable `c` to `"foobar"`.

Awk gives you several builtin functions for performing operations on strings: **length**(*str*) returns the number of characters in *str*; **index**(*s1*,*s2*) finds the first occurence of the string *s2* in *s1*; **substr**(*s*,*m*,*n*) returns an *n* character long substring of *s*, starting at character number *m*; and **split**(*string*, *array*, *separator*) splits *string* up into pieces separated by *separator*, and stores the pieces in *array*. There is also the **getline** function, which reads the next input record, and returns it.

See <undefined> [Input/Output], page <undefined> for more information on **getline**. See <undefined> [Builtin Functions], page <undefined> for detailed descriptions of the rest of these functions.

[What is the operator precedence of concatenation????]

2.5.2 Numeric Operations

Awk provides the usual set of numeric operations. They are named the same as their C counterparts: * for multiplication, / for division, % for remainder (or "mod"), + for addition, and - for subtraction or unary negation. It also provides the *incrementation* operators ++ and --. Since the incrementation operators change the values of variables, they are described in the next section instead of here (see <undefined> [Assignments], page <undefined>).

The arithmetic operators have the usual precedence rules: unary operators have the highest precedence, followed by *, /, and %, with + and - having the lowest. All the binary operators associate to the left. Liberal use of parentheses is always recommended whenever there might be confusion about precedence in your programs.

In **gawk**, the % operator is guaranteed to compute the "trunc-mod" function. This means that `a % b` is computed by finding the quotient `a / b`, truncating it toward zero, and setting `a % b` so that

```
b * int(a / b) + (a % b) == a
```

Using trunc-mod has the desirable effect of making the above equality always hold. It has the not-so-desirable effect that `a % b` can be negative whenever `a` is negative, even if `b` is positive.

Other implementations of **awk** might interpret `%` differently or in a machine dependent fashion, so be careful.

Awk's builtin functions for operating on numbers are `exp`, `log`, `sqrt`, and `int`. `exp(x)` computes the exponential of `x`, `log` the natural logarithm, `sqrt` the square root, and `int` the integer part of `x`, truncated toward 0. See <undefined> [Builtin Functions], page <undefined> for more information on these functions.

2.5.3 Assignments

An *assignment* is a type of expression that sets an *lvalue* to a new value. An lvalue can be either an **awk** variable, an array element (see <undefined> [Variables and Arrays], page <undefined>), or a field (see <undefined> [Fields], page <undefined>). (The name "lvalue" means something that can appear on the left side of an assignment operator.)

lvalue `=` *exp1*
: Stores the value of *exp1* in *lvalue*.

lvalue `+=` *exp1*
: Adds the value of *exp1* to *lvalue*.

lvalue `-=` *exp1*
: Subtracts the value of *exp1* from *lvalue*.

lvalue `*=` *exp1*
: Multiplies the value of *lvalue* by *exp1*.

lvalue `/=` *exp1*
: Divides the value of *lvalue* by *exp1*.

lvalue `%=` *exp1*
: Computes the value of *lvalue* modulo *exp1*, and stores the result in *lvalue*.

Assignment expressions have values, just like any other expressions. Except for the postincrement and postdecrement expressions described below, the value of an assignment expression is identical to the value to which the lvalue is set. Thus,

```
foo = bar = 3
```

computes the value of the expression **bar = 3**, and stores the result into **foo**. This has the effect of setting both **foo** and **bar** to 3, as you would expect. Similarly,

```
foo = (bar += 3) * 2
```

increments the value of **bar** by 3, then stores twice **bar**'s new value into **foo**.

Assignment expressions that perform numerical computations coerce strings into numbers, as usual. See <undefined> [Data Types], page <undefined>.

2.3 6

2.5.3.1 The ++ and -- operators

In addition to the binary assignment operators described above, **awk** also provides post- and pre- increment and decrement operators, just like C. These are described in this section because they store new values into variables and can therefore also be considered "assignment expressions".

lvalue++ This expression causes the contents of *lvalue* to be incremented; i.e., it adds 1 to the number stored in *lvalue*. It has the value that *lvalue* had, *before* the incrementing happened.

++*lvalue* Increments *lvalue* like the above, but has the value that *lvalue* gets *after* the incrementing happens.

lvalue-- Like *lvalue*++, but decrements (subtracts 1 from) *lvalue* instead of incrementing it.

--*lvalue* Like ++*lvalue*, but decrements *lvalue* instead of incrementing it.

2.5.4 Fields

2.5.4.1 Accessing Fields

When **awk** reads an input record, the record is automatically parsed by the interpreter into pieces called *fields*. Fields of the current input line are accessed using the **$** operator. Normally, fields are separated by white space. For example, if your **awk** script has just read the line

```
Frederick Fribbleton Foobar, 29 Bordlesnorf Lane, Cambridge MA 02139
```

then the assignment

```
middle_name = $2
```

will set the **awk** variable **middle_name** to the string **"Fribbleton"**.

The argument to the **$** operator does not have to be a numeric constant. It can be any **awk** expression, which will be coerced to a number using **awk**'s usual coercion rules, then truncated toward 0 to the nearest integer.

The special variable **NF** is set by **awk** when each input line is parsed. Its value is the number of fields that were found in the line. Accessing a field number that is negative or greater than **NF** always returns the empty string. The field **$0** is special; it always refers to the entire input line.

2.5.4.2 Assigning to fields

In **awk**, you can set a field into to a new value in the same way as a variable is set to a new value. Continuing to use the example above, executing the statements

```
$3 = "Flintstone"
$6 = "Blvd"
print $0
```

would cause the string

```
Frederick Fribbleton Flintstone, 29 Bordlesnorf Blvd, Cambridge MA 02139
```

to be printed. Thus, one may think of **$0**, **$1**, and so forth as pseudo-variables whose values are set as each input line is read.

If you set **$0** to a new value, the new value is parsed into fields and the pseudo-variables **$1**, **$2**, ... are re-set accordingly, as is the special variable **NF** (see <undefined> [Accessing Fields], page <undefined>). **[Is this really true?]** 2.5.4.1

2.5.4.3 Specifying how fields are separated

You can change the way **awk** splits the line into fields using the special variable **FS** (for "field separator"). The contents of **FS** is expected to be a one-character-long string. **Awk** scans each input line for that character to determine the positions of fields within the line.

For example, if you set the value of **FS** to the string "," and once again use the input line

```
Frederick Fribbleton Foobar, 29 Bordlesnorf Lane, Cambridge MA 02139
```

from the preceding two sections, then the assignment

```
address = $2
```

will extract the string " 29 Bordlesnorf Lane".

A few values of **FS** are treated specially:

- if **FS** is " " (a space character), then fields are considered to be separated by whitespace, where "whitespace" means any string of one or more spaces and/or tabs. (Other characters such as newline, formfeed, etc. which are considered "whitespace" by other languages are *not* considered whitespace by **awk**.)
- if **FS** contains a newline, ...
- if **FS** is the empty string, ...

The default value of **FS** is a string consisting of a single space.

2.5.4.4 Choosing your Separator Character

Sometimes your input data will contain separator characters that don't separate fields the way you thought they would. For instance, the person's name in the example we've been using might have a title or suffix attached, as in '**Frederick Fribbleton Foobar, LXIX**'. In that case, **address** will be erroneously set to "LXIX"! All you can do to prevent problems like this is choose your data layout and separator characters carefully. Using a character such as '|' instead of ',' as a field separator for this database would have been a good idea.

If **FS** is several characters long, only its first character is used as a field separator. (This may

change in the future). If **FS** is numeric, it is coerced to string as usual before use.

2.6 Conditional Expressions

A *conditional expression* is an expression whose value depends on whether some relation holds between its subexpressions. The value of a conditional expression is always either TRUE or FALSE. Conditional expressions are used in two contexts:

- in *control statements* such as **if**, **while**, and so forth which control the flow of execution of **awk** programs. These statements use conditional expressions to decide what to do. See <undefined> [Control Statements], page <undefined>.
- in *patterns*. Patterns are conditional expressions or pairs of conditional expressions. They control which input lines are processed by a given rule from an **awk** script. See <undefined> [Patterns], page <undefined>.

Here are the different forms conditional expressions can have. In these examples, *exp1* and *exp2* are ordinary (string or numeric) expressions.

First, an ordinary expression can be used as a condition. It is treated as TRUE if the expression evaluates to a nonzero number or a nonempty string, FALSE otherwise.

Next, the usual operations are defined to test for equality and ordering relations between quantities. *Exp1* **==** *exp2* is TRUE if and only if *Exp1* and *exp2* are equal. Similarly, *exp1* **!=** *exp2* is true when *exp1* and *exp2* are *not* equal. There are also **<**, **<=**, **>**, and **>=** for less than, less than or equal, greater than, and greater than or equal, respectively.

In all cases, *exp1* and *exp2* are only compared as numbers if they are both numeric expressions; otherwise, both are coerced into strings and compared lexicographically. See <undefined> [Data Types], page <undefined>.

Besides comparison and the ordering relations, **awk** also provides operations for testing whether regular expressions (or *regexps*) match against strings. (see <undefined> [Regexps], page <undefined> for an explanation of regexps).

regexp TRUE if the input line is matched by *regexp*. *Regexp* is a string describing a regular expression.

exp **~** *regexp*
: True if *exp* (taken as a character string) is matched by *regexp*.

exp **!~** *regexp*
: True if *exp* (taken as a character string) is *not* matched by *regexp*.

Note the appearance of regular expressions in **awk**: they are literal character strings, delimited by slashes, as in '**/Regexp/**'. In the current version of **awk**, regexps must be specified literally, in the **awk** script. They cannot be computed at run time. A future version may change this.

Here is an example of an **awk** fragment that uses a regexp in a conditional expression:

```
# print a message if sentence contains the string "yow"
if (sentence ~ /yow/)
    print "Am I having *fun* yet?"
```

Finally, conditions can be grouped with parentheses, combined using the **&&** and **||** operators, or negated using **!**.

condition1 **&&** *condition2* TRUE if both *condition1* and *condition2* are true. *condition2* is evaluated only if *condition1* is true.

condition1 **||** *condition2*
: TRUE if at least one of *condition1* and *condition2* is true. *condition2* is evaluated only if *condition1* is false.

!*condition* TRUE if *condition* is false.

&& and **||** associate to the left and have lower precedence than the ordering relations. This means that the subexpressions inside the **if** statement in

```
if (age <= 12 || occupation == "programmer")
    ticket_price = 0
```

group correctly (to give free tickets to children and programmers).

3. Control Statements

Awk's control statements are similar to those of C. In all the examples in this chapter, *body* may be either a single statement or a compound statement.

Keywords: if while, for, ..in, break, continue, next, exit.

3.1 The `if` statement

if (*condition*) *body1* [**else** *body2*]

The **if** statement is **awk**'s basic conditional construct. If *condition* is true, *body1* is executed; otherwise, *body2* is executed (assuming that the **else** clause is present). See <undefined> [Conditional Expressions], page <undefined>. Example:

2.6

16

```
if (x % 2 == 0)
    print "x is even"
else
    print "x is odd"
```

3.2 The `while` statement

while (*condition*)
    body

The **while** statement is the simplest looping construct in **awk**. It tests *condition*, and if *condition* is true then it executes *body*. After *body* has been executed, *condition* is tested again and this process is repeated until *condition* is no longer true. Thus, if *condition* is initially false, the body of the loop is never executed. Example:

```
Non-contrived example needed here,
meaning one that doesn't look like a
disassembled for statement.
Suggestions?
```

3.3 The `for` statement

For is another looping statement, but it gives you a fancier control structure than does **while**. It

is similar to the `for` statements that exist in many other languages in that it performs an *increment* every time through the loop. Like in C (but unlike Pascal), what *increment* can do is not limited to adding some constant to an index variable. *Increment* can be any expression, which you must supply.

The basic form of the `for` statement looks like this:

```
for (initialization; condition; increment)
    body
```

This construction executes *initialization*; then, while *condition* is true, it repeatedly executes *body* and *increment*. Thus, it is equivalent to

```
initialization;
while (condition) {
    body
    increment
}
```

except that a `continue` statement inside *body* executes `increment` before re-entering the loop (see <undefined> [Continue Statement], page <undefined>).

The reason there is a `for` statement as well as a `while` statement is that `for` loops are often more convenient to use. When you write a loop, you often want some control variable in the loop to change every time through, and it can be more logical to think of this change as part of the looping construct than as being another statement in the loop body.

Example of a `for` loop:

```
# print odd numbers from 1 to 100
for (i = 1; i <= 100; i += 2)
    print i
```

Note that *initialization* and *increment* are expressions (usually assignment expressions; see <undefined> [Assignments], page <undefined>). Other kinds of statements are not permitted there.

3.3.1 Looping Through an Array

A different form of the **for** statement is used when you want to perform some operation on every element of an array. (see <undefined> [Variables and Arrays], page <undefined>). The construction

```
for (x in array)
    body
```

executes *body* once for each different value that your program has previously used as a subscript of *array*, with *x* set to the value of that subscript. The order in which elements of the array are accessed by this statement is not defined. (It is determined by the hashing mechanism through which array elements are accessed).

If new elements are added to *array* by statements in *body*, one cannot predict whether or not the **for** loop will reach them. Similarly, changing *x* inside the loop can produce strange results. These practices are therefore not recommended.

Here is an example of this form of the **for** statement. It assumes that **frequencies** is an array whose entries represent the number of times each word has been seen in an input text file. See <undefined> [Text Example], page <undefined> for a more detailed example of this type.

```
# Find total number of words that are more than 10 characters long
num_long_words = 0
for (x in frequencies) {
    if (length(x) > 10)
        num_long_words += frequencies[x]
}
```

Note: There is a subtle difference between the way the **for (... in ...)** statement works in **gawk** and the way it works in Un*x **awk** (see <undefined> [Differences], page <undefined>), but it should not affect most programs.

3.4 The **break** statement

The **break** statement jumps out of the innermost **for** or **while** loop that encloses it. Example:

```
# find smallest divisor of num
for (div = 2; div*div <= num; div++)
```

```
        if (num % div == 0)
            break;
    if (num % div == 0)
        printf "Smallest divisor of %d is %d ", num, div
    else
        printf "%d is prime", num
```

3.5 The `continue` statement

Continue, like **break**, is only used inside **for** or **while** loops. It skips over the rest of the loop body so that execution of the loop will go on to the next iteration. It is different from **break** in that **break** jumps out of the loop altogether. Example:

```
# print names that don't contain the string "ignore"
for (x in names) {
    if (x ~ /ignore/)
        continue;
    print x
}
```

Note the use of the **for** ... **in** ... construction in this example (see <undefined> [Array For], page <undefined>).

3.6 The `next` statement

Next forces **awk** to immediately stop processing the current input line and go on to the next one. Thus, no further rules are executed for the current input line. Example:

```
(Example is needed.  Actually, the description
might not even be right.)
```

3.7 The `exit` statement

Exit makes **awk** immediately abort execution of the current rule and stop processing input. This is not the same as making the **awk** process quit immediately, because if your script has an **END** action, that action will still be executed. (If the **exit** statement is encountered *inside* the **END** action, the script exits immediately).

If an argument is supplied to **exit**, its value is used as the return code from the **awk** process.

Example needed again (ho hum).

4. Input and Output

blah blah blah

4.1 Input

When you type an **awk** command to the shell, you can specify one or more *input files.* If you don't specify any input files, **awk** reads all its data from the *standard input*, which is normally your terminal. If you specify input files, **awk** reads data from the first one until it reaches the end; then it reads the second, and so on. Only after the last input file has been exhausted is your **END** action, if you have one, executed. (see <undefined> [Patterns], page <undefined> for more information on **END**).

Your **awk** script can always find out what input file it is currently processing by examining the special variable **FILENAME.** [**Does setting FILENAME open a new input file????**] Input files are always read one record at a time. Normally, a record consists of a line of text, but you can change this by setting the special variable **RS** (for "record separator"). The default value of **RS** is **"\n"** (a newline character).

The special variable **NR** is the number of records that have so far been read from the current input file. It is reset to 0 when a new file is opened. (Is this right???)

4.2 The Print and Printf commands

print [*exp1*, *exp2*, ...] [*redirection*]
: Print *exp1*, *exp2*, ... on the stream specified by *redirection.* If no *redirection* is supplied, **print** prints on the standard output.

printf *format* [*exp1*, *exp2*, ...] [*redirection*]
: Print *exp1*, *exp2*, ... formatted according to *format*, on the stream specified by *redirection* or on the standard output.

The **print** statement prints *exp1*, *exp2*, ... using a standard format which can controlled to some extent using the special variable **OFMT.** See <undefined> [Special Variables], page <undefined>. After printing its arguments it prints a newline.

Because of some strange whim of **awk**'s original designers, a naked **print** statement with no arguments is equivalent to **print $0**. This construction is counterintuitive and we don't recommend its use. None of the examples in this manual use it; but it is documented here in order to prevent it from surprising you.

If you want more precise control over the output format than **print** gives you, use **printf**. With **printf**, you must specify your own *format*, a character string interpreted in the same way as the one used by the C library **printf** function. See <undefined> [C library printf], page <undefined>. **Printf** does not automatically append a newline to its output, so you have to provide one if you want it. Example:

```
printf "%10s %-5d\n", foo, bar
```

prints *foo* and *bar* where *foo* is treated as a character string and *bar* as a number. *Foo* is blank padded on the right to be 10 characters wide. *Bar* is padded to be 5 characters wide, but this time the padding is done on the left because a minus sign was supplied before the field width.

4.2.1 Redirecting output of print and printf

See UNIX (T+Y) p 90

If a *redirection* is supplied to either **print** or **printf**, the output goes to it instead of to the standard output. Redirections are specified in the same way as they are in shell commands:

print foo > *file*
: Print the variable **foo** onto output file *file*. *File* can be any expression. Its value is coerced to a string and then used as a filename.

  The first time *file* is written to in this way, it is truncated to zero length. Subsequent such writes append to *file*.

print foo >> *file*
: Like the above, but on the first write, *file* is not truncated. Instead, the output is appended to it.

print foo | *command*
: Opens a pipe to *command* and writes the value of *foo* to the pipe. [Does gawk wait for outstanding commands to finish before exiting?]

Redirecting output using **>**, **>>**, or **|** asks the system to open a file or pipe only if the particular *file* or *command* you've specified has not already been written to by your program. When a file (or pipe) is actually opened, the filename or command associated with it is cached away by **gawk** and subsequent writes to the same place use the same file descriptor again. The files stay open until

gawk is about to exit. One consequence of this is that the maximum number of files you can write from a single **gawk** run is a few less than the maximum number of files a process can have open at once.

Appendix A. Builtin Functions

The built-in functions of **awk** are mentioned in various places in this manual. They are summarized here for your convenience. There is currently no way in **awk** to define your own functions; however, a future version might provide one. See <undefined> [Improvements], page <undefined> for more information.

In most cases, extra arguments passed to builtin functions are ignored. If an argument is missing, **$0** is passed by default. We consider using this behavior to be sloppy programming practice and don't recommend it, but **gawk** provides it for compatibility with Un*x **awk**.

exp(*x***)** Returns the exponential of *x*, or gets an error if *x* is out of range. The range of values *x* may take on depends on your machine's floating point arithmetic.

log(*x***)** Returns the natural logarithm of *x*, if *x* is positive. Otherwise, gets an error.

sqrt(*x***)** Returns the positive square root of *x*. Gets an error if *x* is negative.

int(*x***)** Returns the integer part of *x*. This is the nearest integer to *x* with absolute value not larger than abs(*x*).

length(*str***)**

Returns the number of characters in *str*. If *str* is a number, the length of the digit string representing that number is returned.

getline Reads the next input record, and returns it. This different from falling through to the next record, which causes the rest of the rules to be interpreted. [Improve this]

index(*s1***,***s2***)**

Finds the first occurence of the string *s2* in *s1*, and returns the offset in *s1* of the beginning of *s2*. 1 represents the first character in *s1*. If *s2* is not found, **index** returns 0.

substr(*s***,***m***,***n***)**

Returns an *n* character long substring of *s*, starting at character number *m*. The first character of a string is character number 1.

split(*string***, *array*, *separator*)**

Splits *string* up into pieces separated by *separator*, and stores the pieces in *array*. The first piece is stored in **vararray[1]**, the second piece in *array***[2]**, and so forth. If the *separator* argument is omitted, the current value of **FS** is used. **[check this]**.

Split splits strings into pieces in a manner similar to the way input lines are split into fields. It returns the number of fields found. See <undefined> [Accessing Fields], page <undefined> for more information on fields.

sprintf(*fmt***,***exp***,...)**

Returns a string containing the values of the argument expressions, formatted according

to the string *fmt*. This works like the C function except it returns its result instead of building it into an output buffer that you pass as an argument.

Appendix B. Special Variables

FS	**FS** is the *input field separator*. This character tells **awk** how to parse lines into fields. Its default value is " " (a string consisting of a single space). See <undefined> [Field Separators], page <undefined>.
NF	**NF** is the number of fields that **awk** found in the current input record. See <undefined> [Accessing Fields], page <undefined>.
RS	This is **awk**'s *record separator*. It is usually a string containing just a newline, which means that an input record consists of a single line of text. See <undefined> [Input/Output], page <undefined>.
NR	This is the number of input records **awk** has seen so far in the current input file. See <undefined> [Input/Output], page <undefined>.
FILENAME	This is the name of the file that **awk** is currently reading from. If **awk** is reading from the standard input, or executing the **END** action of your script, **FILENAME** is the empty string. See <undefined> [Input/Output], page <undefined>. [What about **BEGIN**?]
OFS	output field sep (' ' orig)
ORS	output record sep ('\n' orig)
OFMT	This string is used by **awk** to control how an internal version of the **sprintf** function converts numbers to strings. It is used both by the **print** statement (see <undefined> [Printf], page <undefined>) and for type coercion (see <undefined> [Data Types], page <undefined>). Its default value is **"%.6g"**.

Appendix C. A Text Analysis Example

The following simple example is a complete **awk** program, which prints the number of occurences of each word in its input. It illustrates the associative nature of **awk** arrays by using strings as subscripts. It also demonstrates the **for** *x* **in** *array* construction. Lastly, it shows how **awk** can be used in conjunction with other GNU system tools to do a useful task of some complexity with a minimum of effort. Some explanations follow the program listing.

```
# Print list of word frequencies
{
    for (i = 1; i <= NF; i++)
        freq[$i]++
}

END {
    for (word in freq)
        printf "%s\t%d\n", freq[word], word
}
```

Now the explanations. The first thing to notice about this program is that it has two rules. The first rule, because it has an empty pattern, is executed on every line of the input. It uses **awk**'s field-accessing mechanism (see <undefined> [Fields], page <undefined>) to pick out the individual words from the line, and the special variable **NF** (see <undefined> [Special Variables], page <undefined>) to know how many fields are available. For each input word, the array **freq** is updated to reflect that the word has been seen one more time.

The second rule, because it has the pattern **END**, is not executed until the input file has been exhausted. It prints out the contents of the **freq** table that has been built up inside the first action.

Note that this program has several problems that would prevent it from being useful by itself on real text files:

- Words are detected using the **awk** convention that fields are separated by whitespace and that other characters in the input (except newlines) don't have any special meaning to **awk**. This means that punctuation will count as part of words.
- **Awk** considers upper and lower case characters to be distinct. Therefore, '`foo`' and '`Foo`' will not be treated by this program as the same word. This is undesirable since in normal text, words are capitalized if they begin sentences, and a frequency analyzer should not be sensitive to that.
- The output does not come out in any useful order. You're more likely to be interested in which

words occur the most frequently, or in having an alphebetized table of how frequently each word occurs.

The way to solve these problems is to use other GNU utilities to process the input and output of the **awk** script. Suppose the script shown above is saved in the file '**frequency.awk**'. Then the shell command

```
tr A-Z a-z < file1 | \
  tr -cd 'a-z\012' | \
  awk -f frequency.awk | \
  sort +1 -nr
```

produces a table of the words appearing in '**file1**' in order of decreasing frequency. The first two **tr** commands in this pipeline respectively translate all the upper case characters in '**file1**' to lower case, and delete all the characters in the input to **awk** except for lower case characters and newlines. (The second argument to the second **tr** is quoted to protect the backslash in it from being interpreted by the shell.) **Awk** then reads this suitably massaged data and produces a word frequency table which is in no particular order. Finally, the **awk** script's output is sorted by the **sort** command and printed on the terminal. The options given to **sort** in this example specify that the first field of **sort**'s input should be skipped over, that the sort keys should be treated as numeric quantities (otherwise '**15**' would come before '**5**'), and that the sorting should be done in descending ("reverse") order.

See the documentation for the **tr** and **sort** commands for more information on how to use them.

C.1 More examples

*** more examples are needed

Appendix D. Implementation notes and bugs

This appendix contains information mainly of interest to implementers and maintainers of **gawk**. Everything in it applies specifically to **gawk**, and not other implementations. Newly discovered bugs and incompatibilities beteen **gawk** and Un*x **awk** should be either fixed or documented in this section.

D.1 Known bugs and misfeatures in gawk.

Surely there are some.

D.2 How gawk works inside

yacc parser and simple recursive evaluator. regexps are compiled at parse time and the compiled regexps are cached. How arrays, file redirects, etc. are implemented as hash tables. describe grody yylex hacks to handle concatenation operator, etc. Also mention how field ptrs are cached.

D.3 Improvements: suggestions for future improvements

Here are some projects that would-be **gawk** hackers might like to take on. They vary in size from a few days to a few weeks of programming, depending on which one you choose and how fast a programmer you are. Please send any improvements you write to the maintainers at the GNU project.

1. Add v8 features: The version of **awk** used in Eighth Edition Un*x has several new features, such as user defined functions, runtime regular expressions, file manipulation functions, etc. They are described in [get exact reference]. It should not be too hard to update **gawk** (and this manual, of course) to include all these new features.
2. State machine regexp matcher: At present, **gawk** uses the backtracking regular expression matcher from the GNU subroutine library (see <undefined> [Regexps], page <undefined>). If a regexp is really going to be used a lot of times, it is faster to convert it once to a description of a finite state machine, then run a routine simulating that machine every time you want to match the regexp. This is how scanners generated by programs like **Lex** work.

A description of the algorithms used for doing this kind of thing can be found, for example, in *Compilers: Principles, Techniques, and Tools* by A. V. Aho, R. Sethi, and J. D. Ullman.

3. **Awk** script compilation: **gawk** uses a **yacc** parser to convert the script given it into a syntax tree; the syntax tree is then executed by a simple recursive evaluator. Both of these steps incur a lot of overhead, since parsing can be slow (especially if you also do the previous project and convert regular expressions to finite state machines at compile time) and the recursive evaluator performs many procedure calls to do even the simplest things.

   It should be possible for **gawk** to convert the script's parse tree into a C program which the user would then compile, using the normal C compiler and a special **awk** library to provide all the needed functions (regexps, fields, associative arrays, type coercion, etc.).

   An easier possibility might be for an intermediate phase of **gawk** to convert the parse tree into a linear byte code form like the one used in GNU Emacs Lisp. The recursive evaluator would then be replaced by a straight line byte code interpreter that would be intermediate in speed between running a compiled program and doing what **gawk** does now.

Please keep in mind that from the point of view of the GNU project, none of these improvements are terribly valuable. The v8 extensions are not needed for compatibility reasons because AT&T is not distributing Eighth Edition Un*x to anyone except a few universities who have signed very confining non-disclosure agreements with them. The versions of **awk** distributed with BSD Un*x and System V do not include the new features. If they start doing so, then adding them to **gawk** will become more important. Otherwise, it's not vital for GNU to have a more powerful **awk** since we have **ICON**. The other improvements primarily make **gawk** run faster.

D.4 Differences: differences from Unix awk

Array elements should be created only when set, not when examined. [bug?] an object is a number or a string, not both. Thus 1e3!=1000 maybe. Maybe a node commenting on Pierce's paper and saying which bugs are fixed.

Appendix E. Glossary

Action
: A series of **awk** statements attached to a rule *(q.v.)*. If the rule's pattern matches an input record, the **awk** executes the rule's action. See <undefined> [Actions], page <undefined>.

Amazing Awk Assembler
: Henry Spencer at the University of Toronto wrote a retargetable assembler completely as **awk** scripts. It is thousands of lines long, including machine descriptions for several 8-bit microcomputers. It's not relevant to this manual but is such a feat of hacking that I wanted to mention it here. It is distributed with **gawk** and is a good example of a program that could better have been written in another language. See <undefined> [Using Awk], page <undefined>.

Assignment
: An **awk** expression which changes the value of some **awk** variable or data object. An object that you can assign to is called an *lvalue (q.v.)*. See <undefined> [Assignments], page <undefined>.

Awk
: The **awk** language, as opposed to any particular implementation of it.

Awk script
: Another name for an **awk** program. An **awk** program is a series of *rules (q.v.)*; for each input record given to the program, the program's rules are all processed in turn.

Builtin Function
: **Awk** provides built in functions that perform various numerical and string computations. Examples are **sqrt** (for the square root of a number) and **substr** (for a substring of a string). See <undefined> [Builtin Functions], page <undefined>.

C
: The system programming language that most of GNU is written in. **Awk** has C-flavored syntax, and this manual points out similarities between **awk** and C when appropriate. However, the manual tries to be understandable by people not familiar with C.

Compound Statement
: A series of **awk** statements, enclosed in curly braces. Compound statements may be nested.

Concatenation
: Concatenating two strings means sticking them together, one after another, giving a new string. Thus, the string '**foo**' concatenated with the string '**bar**' is the string '**foobar**'.

Conditional Expression
: A relation which is either true or false, such as **(a < b)**. Conditional expressions are used in **if** and **while** affect the control flow of **awk** programs, and in patterns *(q.v.)* to select which input records to process.

Curly Braces

The characters '{' and '}'. Curly braces are used in **awk** for grouping compound statements and for delimiting actions.

Data Objects

Awk data objects include numbers and strings. Numbers are converted into strings and vice versa, as needed. See <undefined> [Data Objects], page <undefined>.

Field — When **awk** reads an input line, it splits the line into pieces separated by whitespace (or by a separator character which you can change by setting the special variable **FS**). Such pieces are called *fields*. See <undefined> [Fields], page <undefined>.

Format — Format strings are used to control the appearance of output of the **printf** statement. Also, data conversions from numbers to strings are controlled by the format string contained in the special variable **OFMT**. See <undefined> [Printf], page <undefined>.

Gawk — The GNU implementation of **awk**, written by Paul Rubin and Jay Fenlason.

Icon (string processing language)

A powerful string and pattern processing language written at the University of Arizona, as a successor to SNOBOL4.

Input Record

A single chunk of data read in by **awk**. Usually, an **awk** input record consists of one line of text.

Keyword — **Awk**'s keywords are **if**, **while**, **for**, ..**in**, **break**, **continue**, **next**, and **exit**. See <undefined> [Control Statements], page <undefined>.

Lvalue — An expression that can appear on the left side of an assignment operator. In most languages, lvalues can be variables or array elements. In **awk**, a field designator can also be used as an lvalue. See <undefined> [Assignments], page <undefined>.

Number — A numeric valued data object *(q.v.)*. **Gawk** normaly uses double precision floating point to represent numbers. See <undefined> [Data Types], page <undefined>.

Pattern — Patterns tell **awk** which input records are interesting to which rules. A pattern is an arbitrary conditional expression *(q.v.)* against which an input line is tested. If the condition is satisfied, the pattern is said to *match* the input record. A typical pattern might compare the input record against a regexp *(q.v.)*.

Range (of input lines)

A sequence of consecutive lines from the input file, that starts at an arbitrary point and ends at another arbitrary point. A pattern can specify ranges of input lines for **awk** to process, or it can specify single lines. See <undefined> [Patterns], page <undefined> for more information about patterns and ranges.

Redirection

You can redirect the output of the **print** and **printf** statements to a file or a system command, using the **>**, **>>**, and **|** operators. See <undefined> [Output Redirection], page <undefined>.

Regexp — Short for *regular expression*. A regexp is a pattern that denotes a set of strings, possibly an infinite set. For example, the regexp '`R.*xp`' matches any string starting with the letter `R` and ending with the letters `xp`. In **awk**, regexps are used in patterns and in conditional expressions. See <undefined> [Regexps], page <undefined>.

Rule — A segment of an **awk** program, that specifies how to process single input lines. A rule consists of a *pattern* and an *action (q.v.)*. **Awk** reads an input record; then, for each rule, if the input record satisfies the rule's *pattern*, **awk** executes the rule's *action*. Otherwise, the rule does nothing for that input record.

SNOBOL — A somewhat abstruse string processing language written in the 1960's ("StriNg Oriented symBOlic Language"). It had extremely powerful pattern matching facilities but only rudimentary syntax and control structures. It has supposedly been superseded by `Icon` *(q.v.)*. SNOBOL4 was its hairiest incarnation.

Special Variable — The variables `FS`, `NF`, `RS`, `NR`, `FILENAME`, `OFS`, `OR`, and `OFMT` have special meaning to **awk**. Changing them affects **awk**'s running environment. See <undefined> [Special Variables], page <undefined>.

Stream Editor — A program which reads records from an input stream and processes them one (or several) at a time. This is in contrast with normal batch programs (which may expect to read their input files in entirety before starting to do anything), and with interactive programs (which require input from the user).

String — An arbitrary sequence of characters, such as `"I am a string"`. See *data objects*.

Un*x **awk** — The version of **awk** shipped with Un*x (a GNU-like operating system marketed by AT&T).

Whitespace — A sequence of one or more blank or tab characters occuring inside an input record or a string. Fields are normally separated by white space.

Awk Script An Awk program p. 3

sprintf The sprintf function formats the function's output before printing it to memory. See App A, Builtin Functions, p. 24